Spring 2015

A Course Module On Application Logic Flaws

Lindsay Simpkins
North Carolina Agricultural and Technical State University

A Course Module on Application Logic Flaws

Lindsay Simpkins

North Carolina A&T State University

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Department: Computer Science

Major: Computer Science

Major Professor: Dr. Xiaohong Yuan

Greensboro, North Carolina

2015

The Graduate School
North Carolina Agricultural and Technical State University

This is to certify that the Master's Thesis of

Lindsay Simpkins

has met the thesis requirements of
North Carolina Agricultural and Technical State University

Greensboro, North Carolina
2015

Approved by:

Dr. Xiaohong Yuan Major Professor	Dr. Anna Huiming Yu Committee Member
Dr. Kenneth Williams Committee Member	Dr. Gerry Vernon Dozier Department Chair

Dr. Sanjiv Sarin
Dean, The Graduate School

Biographical Sketch

Lindsay Simpkins obtained her Bachelor of Arts degree at Guilford College, double majoring in Mathematics and Art. After working retail for several years, she decided to find something more challenging and engaging by pursuing a Master of Science in Computer Science at North Carolina A&T State University, with a focus in Secure Software Engineering. During her time at NC A&T State University, Lindsay has been both a Teaching Assistant and a Research Assistant. Her research has led to the acceptance of one conference paper, publication of one journal article, and several poster presentations. She was also accepted to the Upsilon Pi Epsilon and Phi Kappa Phi honor societies.

Dedication

This thesis is dedicated to my husband, my family, and my best friend.

Acknowledgments

I would like to thank Dr. Anna Yu for accepting me into the Master of Science in Computer Science program at North Carolina A&T State University, and making it possible for me to attend full-time because of a scholarship.

I would also like to thank Dr. Kenneth Williams for his information and enthusiasm about the Computer Science program when I was a prospective student, and his guidance on course selection.

I would also like to thank my advisor, Dr. Xiaohong Yuan, for the opportunity to do research with her, and for her wonderful feedback on my research, publications, and thesis.

Finally, I would like to thank my husband for all of his support, and my family for their encouragement and help with editing this thesis.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ACM – Association for Computing Machinery

API – Application Programming Interface

CAPEC – Common Attack Pattern Enumeration and Classification

CERT – computer Emergency Readiness Team

COMP – Computer Science Department

CWE – Common Weakness Enumeration

HTML – Hypertext Markup Language

HTTP – Hypertext Transfer Protocol

IAS – Information Assurance and Security

IBM – International Business Machines Corporation

ID – Identification

IEEE – Institute of Electrical and Electronics Engineers

IRB – Institutional Review Board

KLOC – Thousand Lines of Code

MITRE- The MITRE Corporation

OWASP – Open Web Application Security Project

SEI – Software Engineering Institute

VC – Verification Condition

WASC – Web Application Security Consortium

Abstract

Software security is extremely important, and even thoroughly tested code may still have exploitable vulnerabilities. Some of these vulnerabilities are caused by logic flaws. Due to the nature of application or business logic, few automated tools can test for these types of security issues. Therefore, it is important for students to learn how to reduce the number of logic flaws when developing software, and how to test for them manually. A course module with a case study was created to teach students about this topic. Case-based teaching methods are used because it allows students to better apply learned skills to real world industrial settings, and there is a lack of case studies available for current software engineering curriculum.

The course module includes an introduction, a quiz on the reading, an animated PowerPoint about the case, and a set of discussion questions. The introduction covers what logic flaws are, reducing logic flaws during software development, and how to test for them manually. The case is about eCommerce merchant software Bigcommerce using PayPal Express to collect payment. A flaw lets attackers complete an expensive order using the payment intended for a cheaper order. An animation was created to trace the HTTP interactions and back-end code representing the steps of the exploit from this case, and explain the manual testing method used to discover the exploit. A set of discussion questions has students apply this method to similar code, to find potential vulnerabilities and then fix them.

This course module was taught in COMP 727 Secure Software Engineering at North Carolina A&T State University in the Spring 2015 semester. A pre-survey and post-survey on the learning objectives shows students felt they improved their knowledge and skills relating to application logic flaws. A quiz based on the reading shows students understood the material. The quality of student discussions was very high. Discussion question results were graded using a

rubric, and three-quarters of the class received an 85% grade or higher. Overall, this case study was effective at teaching students about application logic flaws. It will be made available to other universities, and can be easily integrated into existing curriculum.

CHAPTER 1

Introduction

This chapter introduces the topic of application logic flaws, the course module on application logic flaws developed in this research, and the contributions and limitations of this research.

1.1 Application Logic Flaws

Software that is thoroughly tested for security vulnerabilities still has an estimated 5 vulnerabilities per Thousand Lines of Code (KLOC) [1, 2]. If the software is only tested for functionality, and not security, that number is closer to 50 vulnerabilities per KLOC [1-3]. This has an enormous impact when the code base may be millions of lines long. Windows 7, for example, was reported to have approximately 40 million lines of code in 2009 [4].

Vulnerabilities are caused by defects in code. Security expert Gary McGraw separates these into two categories: bugs, and flaws [1, 2]. Bugs are defined as implementation problems in software, such as an off-by-one error, buffer overflows, and using unsafe methods. Bugs will only exist at the code level, can be discovered using static analysis scanning tools, and can typically be fixed on one line or in a localized area. Flaws, on the other hand, are issues that exist at both the code level and the design and software architecture level. Examples include poor access control, insufficient process validation, lack of encryption, and bad error handling (fail-open) [2, 5, 6]. Flaws are typically related to application and business logic, or the architectural design. There could be problems with the logic itself, or with the code that implements the logic. To fix flaws, the logic or software architecture may need to be redesigned, and it could affect multiple areas of code.

An estimated 50% of software security vulnerabilities are flaws [1, 2]. For example, there could be an estimated 100,000 to 1 million flaws in Windows 7, which controls just over half the global market share of desktop operating systems [7]. These software security issues have the potential to adversely impact a huge number of individuals, businesses, and government.

Logic flaws encompass a range of vulnerabilities relating to privilege manipulation and transaction control manipulation [8]. Most static analysis tools use rules based on defined patterns, and are effective at identifying bugs. Several automated security testing tools, such as Fortify [9] or AppScan Source Edition [10], can trace tainted input through applications' components. The scan results show the data flow path that the tainted input takes through the code, from the source (user controlled-input) to the sink (a dangerous function call or operation). These types of static analysis scanners can be used to check for vulnerabilities related to malicious input, such as poor input validation flaws, or bugs such as unsafe methods. Application logic, however, is inconsistent and does not follow defined patterns. For example, one web application may implement the majority of authentication logic on the server side, while another places it on the client side. It is important to note that logic on the client side is extremely bad practice [11], but still occurs frequently[1]. Testing for logic flaws also depends on the type of exploit [12]. Since there are no well-developed automated tools available to identify logic flaws, it must be done manually [2, 8, 12].

At Blackhat Europe 2011, Los and Jagdale asserted that "*Logic* implies human thought required" [8]. It is important for computer science students to learn how to reduce the number of

[1] CWE-603: Use of Client-Side Authentication is a child of CWE-287: Improper Authentication, which maps to two entries on OWASPs' 2013 Top 10 List: A2-Broken Authentication and Session Management (Widespread Prevalence), and A4-Insecure Direct Object Reference (Common Prevalence).

logic flaws when developing software using recommended industry standard best practices, and how to identify and mitigate flaws in existing software. Testing software early in the development lifecycle, as opposed to late in development, can also significantly reduce the cost and time it takes to fix vulnerabilities [2, 13-16], especially if the software architecture or logic has to be redesigned. In addition to refactoring the design or architecture, there would likely need to be new code development, software testing, deployment, and more, all of which increase the cost to mitigate the vulnerability [17]. Research by Shull et al., with support from companies such as IBM and Lockheed Martin, found that "Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase." [15]. The Microsoft Security Response Center estimated each security issue requiring a security bulletin cost around $100,000 [17].

However, the topic of application logic flaws seems to be poorly addressed or omitted in both software engineering courses and textbooks. A course module with a case study was developed to teach students about this topic. Case-based teaching methods are used because they allow students to better apply learned skills to real world industrial settings. They can bridge the gap that exists in current software engineering education between theoretical concepts and industry-related skills [18]. There is also a need to develop case studies that are few in the current software engineering curriculum [19].

1.2 A Course Module on Application Logic Flaws

The course module has a document that introduces the topic of logic flaws and explains where logic flaws come from, how to reduce logic flaws during development, and adding logic flaw testing alongside functional testing. It also describes the manual code review method used

to discover an exploitable logic flaw in the case being studied. A short quiz based on the required reading helps ensure the introduction was read and understood.

The case used here is the eCommerce software Bigcommerce (previously known as Interspire Shopping Cart) when it used the third-party cashier PayPal Express to collect payment, presented in the research article *How to Shop for Free Online: Security Analysis of Cashier-as-a-Service Based Web Stores* [20]. Using a manual code review method, and treating PayPal Express as a black box where only the inbound and outbound messages could be observed due to closed-source code, an exploit was discovered in Bigcommerce where an attacker could use one session with a successful payment status to pay for a different order created in a second session. It allowed the attacker to complete an expensive order using the payment intended for a cheaper order.

This case study covers the logic involved in maintaining state between three parties, maintaining state in an eCommerce transaction, and enforcement of transaction workflow. It is a simple but powerful illustration on how a logic flaw could cause an exploitable security vulnerability. The cause of the logic flaw was that the order ID number was not bound to the session ID number, and a digitally signed argument containing information about the order was still created and made accessible to the attacker even if payment failed.

A PowerPoint animation (Figures 1, 2) was created to trace through the HTTP interactions and back-end code representing the steps of the exploit from the case, and help explain the manual testing methods used to discover the exploit. A set of discussion questions was developed based on Blooms Taxonomy [21] to have students apply this testing method to similar code in order to find and fix potential vulnerabilities. The effectiveness of the coursemodule was evaluated through pre- and post-surveys rating students' knowledge level,

student responses to the discussion questions using a rubric for grading, and the quality of discussion results.

The course module was taught in the COMP 727 Secure Software Engineering course at North Carolina A&T State University in the Spring 2015 semester.

1.3 Contributions and Limitations

This course module is meant to be easily integrated into an existing upper-level undergraduate or graduate computer science course, and can be used either in-class or online. The course module should have 2 lecture hours and 1 hour for student discussion (or equivalent of one week instruction). It fits under the Software Engineering Knowledge Area in the *Computer Science Curricula 2013* written by ACM and IEEE [22], specifically the topic of Software Verification and Validation, if being used for an upper-level undergraduate course. It can also be taught as part of the Secure Software Engineering topic in the Information Assurance and Security (IAS) Knowledge Area. In the *Software Assurance Curriculum Project* [23] for undergraduate courses by CERT Software Engineering Institute (SEI) at Carnegie Mellon University, the course module could be used in Software Security Engineering courses for the topics of secure software architecture and design or secure coding and testing, or in Software Quality Assurance courses for testing methods topic. For graduate-level courses, this course module could be used under the core body of knowledge areas 6.2.1 Development methods and 6.3.5 Testing for assurance [24]. Software engineering or security testing courses that use *Building Security In* [2] as a textbook can use this course module after the Architectural Risk Analysis chapter.

The limitation of evaluating this course module is that it only includes results from only one university. While the results were promising, they are from a sample that was too small for significant statistical analysis results.

The case being used is a web-based application, and students may have difficulties applying this manual testing method to software that does not have client-server interactions, as the code structure may be different. It is also about an eCommerce transaction, and students may have difficulties applying this manual testing method to other types of software, as the process workflow may be different.

The rest of the thesis is organized as follows: Chapter 2 provides a literature review of using case studies in computer science education, existing resources on application and business logic flaws, and prototypes of automated testing for logic flaws. Chapter 3 describes the materials developed for the course module on application logic flaws. Chapter 4 describes the results and teaching experience. Chapter 5 concludes the thesis and discusses future work.

CHAPTER 2

Literature Review

This chapter covers the use of case studies as a teaching method, existing resources on application and business logic flaws, prototypes for automated testing of logic flaws, and information about the case used in this course module.

2.1 Using Case Studies for Teaching

Case studies are effective teaching tools, and widely used in disciplines such as business, law, and medicine [25-28]. Engineering fields including Mechanical Engineering, Civil Engineering, and Chemical Engineering [29-32] use case studies as well. Computer Science has more recently adopted case studies into university level education. For example, use of case studies was mentioned several times in the ACM/IEEE Computer Science Curricula 2013 [22] for teaching ethics in computing, the effects of computing on a variety of populations, causes of computer failures, parallel programming, and distributed systems. However, in the 2008 interim revision [33], only one case study was mentioned, which was on object recognition and tracking for graphic and visual computing. The computer science field lacks a breadth and depth of relevant case studies [19]; most commercially available Harvard-style case studies are focused on business or management, and cases designed for Information Systems or Information Technology still lack necessary technical details [19]. As security in general lacks coverage within computer science, it is reasonable to assume there is also an inadequate number of software security focused case studies available.

Research has shown that case studies are an important and effective teaching method that engages students in active learning, help students develop skills such as problem solving and working as part of a team, and increase students' motivation to learn since the topic is more

enjoyable [18, 34-37]. Benefits of case studies in university education include providing a real-world context to the topic being covered, requiring students to apply critical thinking and analysis, and helping students synthesize the content of the course [38]. Daun et al. noted that "Even theoretical disciplines are beginning to adopt more experience-oriented instruction as opposed to passive, lecture-oriented instruction." [39].

The computer science field also suffers from a gap between academic theory and industry skills and best practices [39-41]. Computer science students may understand the examples used in academia, but have little knowledge of challenges and practical aspects faced by industry [39, 42]. Garg [43] and Wohlin [44] found that additional instruction was needed before students could apply skills learned in university courses to industry settings. A multi-year study [39] showed that using industry-based case studies in a graduate-level requirements engineering course resulted in higher quality discussions, more in-class participation, higher exam results, and higher self-reported levels of positive learning experience, compared to a more traditional approach for teaching the course. Students also self-reported higher levels of problem solving skills, ability to work independently, and interest in the course, compared to other courses in the department [39]. Several case studies on information security topics have also been taught at the undergraduate and graduate level with successful results [23, 34-35, 45-46]. Students reported they were confident about applying the learned knowledge to future jobs [45].

2.2 Resources on Application and Business Logic Flaws

Flaws may be present in the business or application logic, or in the software architecture. Flaws related to software architecture, such as badly defined trust boundaries, may be difficult to identify without proper knowledge and an accurate overview of the system, and can be found by performing an architectural risk analysis [2]. Flaws related to logic can be very subtle, and are

typically found by manually testing and tracing manipulated data and manipulated data flows through the entire process [4, 12, 20].

Several organizations maintain updated databases of different types of business or application logic flaws. A taxonomy of business logic vulnerabilities is available in the Common Weakness Enumeration (CWE) by MITRE, a non-profit organization that operates federally funded research and development centers [47]. This database also maps to other software security databases, such as the Common Attack Pattern Enumeration and Classification (CAPEC) database also operated by MITRE [48]; the Threat Classification database created by Web Application Security Consortium (WASC), a non-profit comprised of security experts and industry practitioners working to develop standards for web application security; CERT Secure Coding Standards by SEI; and several others. The non-profit Open Web Application Security Project (OWASP) also lists categories of flaws to address during testing [12] and a *Business Logic Security Cheat Sheet* [49]. Table A-1 in Appendix A [50] cross-links OWASPs' *2013 Top 10 List* [51] of web application weaknesses with *2011 CWE/SANS Top 25 Most Dangerous Software Errors* [52], and categorizes them into bugs and flaws. Seven out of the Top 10 web application weaknesses and 16 out of the Top 25 software errors are flaws.

Several resources provide specific examples of logic flaw exploits and explain how they could be prevented during software development, but do not cover testing for logic flaws in existing software. *Seven Business Logic Flaws That Put Your Website At Risk* [6] walks through seven real-world attacks to break down how the applications were exploited and how this could have been prevented. *The Web Application Hacker's Handbook* [5] reviews eleven exploits and their prevention methods relating to poor input validation, circumvention of workflow, information leakage, and abuse of functionality. *Top 10 Business Logic Attack Vectors* [53] lists

ten logic flaw exploits using a sample application, and how to test for those flaws. However, these recommendations are fairly specific to the types of flaws or the type of application. For example, do not display full usernames on an online auction website, since an attacker can take advantage of the mechanism that locks accounts when multiple login attempts are made with an incorrect password, to lock competing bidders out of their accounts in order to win auctions.

Recommendations for preventing logic flaws during software development are more general than the recommendations for testing, and are covered in *Building Security In* [2] and OWASP's *Business Logic Security Cheat Sheet* [49]. These industry standard best practices are summarized as follows, though this is not an all-inclusive list by any means.

- Use detailed and thorough software requirements for both functionality and security [2]; security will be incorporated into the software from the start, and functional and security testing are based on the requirements [54].
- In addition to defining use cases during the requirements phase of software development, misuse and abuse cases should be defined as well [2, 49]. This helps explain what the software shouldn't do, how it should react to attacks or failures, defines how the software should respond to abnormal input or manipulated data flows, and may be used for some types of security testing [2].
- During development, the design of the application should be reviewed. Assumptions made by the application should be challenged and the dependencies and interactions between different components should be checked. Documentation of the software should be detailed, thorough, and up-to-date [2, 5], especially for the business logic, design, and software architecture.

- Risks should be tracked throughout the development lifecycle, and may be used for risk-based security testing [2].
- All input, from either the user or another server, should be verified [5].
- Finally, if the software is an online web application, the server should control the application logic, not the client side, or it may be bypassed by an attacker [5, 8].

2.3 Automated Testing Prototypes

There are several prototypes developed to automate the process of finding logic flaws, however they have many limitations. Unlike bugs, flaws don't have a common 'signature' to scan for [5]. Humans can identify behavioral patterns better than software tools, and in these prototypes human input is still needed for extracting the logical model of the application or verifying the results. As with traditional automated security scanning tools, there can be false positives and false negatives, therefore scan results cannot be fully relied on to ensure the software is free of vulnerabilities [5, 55]. Automation generally relies on patterns, but logic may be unique to an application. It is possible that tools could be developed where the custom business logic of an application would be input during setup, and then the tool performs repeatable testing [8].

Pellegrino and Balzarotti [56] state that only general logic vulnerabilities may be found through existing automated tools, and vulnerabilities specific to a particular application must still be found through manual inspection. They used an automated black-box approach by monitoring network traces when users interact with an application, and then extracting behavioral patterns. The goal was not to create an accurate model of the application, but to show a simple model of the application logic was sufficient to identify logic flaws during automated reasoning. The prototype did not require the source code of the application. It was successful at

identifying flaws, but has several limitations. First, it is focused only on the classes of logic flaws found in eCommerce applications, and does not cover instances where the attacker is a malicious merchant or the attacker can tamper with messages sent between the merchant and payment service. The final results from the tool still needed manual review to distinguish between presentation issues, where the internal state is correct but displayed data is not, and cases where the internal state is not correct [56].

Felmetsger et al. [57] developed a prototype called Waler that monitors normal execution of Java servlet-based web applications, and infers relationships between variables. The goal was to identify the specifications of the application without human feedback. Limitations include the need to monitor real users interacting with the application, though it may be possible to use a scripting tool such as Selenium [58] to automate simulated user browsing activity. Waler relies on verifications already implemented in the applications' code; this means that if the application never verifies that a number input by the user is negative when it should be positive, Waler assumes both positive and negative numbers are permissible. Currently, an invariant detector called Daikon [59] is used to derive the program specifications from the data gathered by Waler. Therefore, Waler is bound by the scalability issues of Daikon, which has trouble scanning large trace files and may run out of memory if too many variables are checked or too many invariants created [60]. Waler is also only able to scan Java servlet-based web applications [57].

After discovering several logic flaws using a manual code review method, Wang et al. used a tool called Corral [61] (previously known as Poirot) developed by the Microsoft Research Team to scan for application logic flaws. The goal was to check if Corral could find these flaws using a model of the logic in Bigcommerce; it found all the flaws they discovered with manual testing as well as several others. Corral can be used on C, .NET, or Java programs and "…uses

bounded goal-directed symbolic search techniques to find assertion violations" [62]. Wang et al. manually extracted the logic of Bigcommerce and PayPal Express into a C header file containing implemented variables and methods, and created a symbolic driver program in C. The symbolic driver called five API methods in a nondeterministic order, using nondeterministic argument values, and completed by calling the final checkout method for Bigcommerce. Corral generated a verification condition (VC) based on whether there was a correct payment record corresponding to a completed order. A correct order payment meant the payment amount matched the total price of the order, the merchant was the receiver of payment, and the payment was intended for that order specifically. The VC was then verified by the theorem prover Z3 [63]. Corral could loop through the symbolic driver program multiple times, and the number of loops it was set to run bounded the number of API method calls made during execution.

Limitations of Corral include only accepting C, .NET, and Java source code, and the need for developing a driver program with assertions designed for use with Corral. Installation requires several other programs, such as Z3, and an intermediate program used to convert the source code to an Intermediate Verification Language called Boogie. The intermediate program varies depending on the source code language, and may have its own installation requirements. Wang et al. encountered an out-of-memory error if Corral was looped more than six times on their symbolic driver program, and it also took up to four hours to run six loops [20]. Despite the limitations, Wang et al. said automated verification is necessary; when looping six times through the symbolic diver, there were hundreds of thousands of times Corral had to backtrack to where a decision was made previously and, branch to another data flow path, which is unrealistic for manual testing [20].

2.4 The Case Study

The article *How to Shop for Free Online* [20] covered two popular merchant softwares: NopCommerce and Bigcommerce, and four cashier services: PayPal Standard, PayPal Express, Google Checkout, and Amazon Payments. Serious logic flaws were discovered in NopCommerce and Bigcommerce with how they integrated PayPal Standard, PayPal Express, or Google Checkout to collect payments, and with Amazon Payments itself. These logic flaws allowed an attacker to complete an order for free or at a reduced price, use a replay attack to complete an order of the same price multiple times but only pay for the first order, or complete additional orders for free after paying for one small order.

Wang et al. found at least one serious logic flaw in each case they reviewed, using a manual testing method described below. They then verified these flaws using Corral, as mentioned in Section 2.3. The flaw for Bigcommerce and PayPal Express involved using two sessions, and replacing the digitally signed argument from one session with one in a second session. The attacker could pay for a cheap order, but stop the final step of the checkout process from completing. The first session now has a stored payment record associated with it. Then, the attacker creates an expensive order in a second session, but skips the payment step. A digitally signed variable containing information about the expensive order is still generated by Bigcommerce, even though the checkout fails. Finally, the signed argument from the second session is used to replace the signed argument in the first session, and the final step of the first checkout process is resumed. Thus, the expensive order is completed with the payment intended for the cheaper order.

The testing method used by Wang. et al. was to:

1) Manipulate the data. Identify all arguments used by the API methods in the web application. If an argument is unsigned, test unexpected values such as manually changing the value during the checkout process and leaving the argument value empty. If an argument is digitally signed, test reusing the argument in a replay attack, replacing it with the same signed argument generated from a second session, and leaving the argument value empty.
2) Manipulate the workflow with expected user input. Try skipping one or more steps of the checkout process, and try running the steps in an incorrect order.
3) Manipulate the workflow with data manipulations from step 1.

For each of the steps 1-3, manually trace through the entire checkout process to identify how each argument affects the internal state of the application. Additional tests for step 1 not used by Wang et al. would be to remove the whole argument from the HTTP request, or add extra arguments to the HTTP request for arguments used at a later step in the checkout process, since attacker can manually add them to the HTTP request at an earlier step [5].

Wang et al. also created a case study based on their research [64]. The C header file mentioned in Section 2.3 was provided, along with a template C driver program. Students must sift through the header file to discover the logic flaws for each of the four cashier services, either using the manual code review method, or by following the logic flaw outlined in *How to Shop for Free Online: Security Analysis of Cashier-as-a-Service Based Web Stores* [20], and complete the driver program. Variables used by the attacker would be given values and then methods from the C header file would be called in the right order to exploit the logic flaw, using the attacker-accessible variables as the method arguments. One solution was given as an example. The symbolic C driver program for Corral, mentioned in Section 2.3, was also provided along with an installation of Corral.

CHAPTER 3

The Course Module on Logic Flaws

This chapter discusses the learning objectives of this course module, the rationale for developing this course module, the materials that were created, and the methodology for evaluating this course module.

3.1 Learning Objectives

Logic flaws are consistently among the primary causes of commonly exploited software vulnerabilities [2, 14, 51, 52]. Since logic flaws are issues at both the code and design and software architecture level, computer science students must learn to move beyond the traditional software security paradigm, and understand how to test input validation in combination with workflow manipulation. Application logic does not follow defined, repeatable patterns, which makes it a poor candidate for automated testing tools. However, there are existing manual testing methods specifically used to identify logic flaws, as discussed in Sections 2.2 and 2.4

The goal of this course module is to help computer science students understand the concept of logic flaws and why they must be tested for manually. They should also be able to identify, critique, and then fix logic flaws in the source code provided in the discussion questions. To guide the development of this course module, a set of learning objectives (Table 1) was created. The discussion questions are directly related to the learning objectives, in order to evaluate if the learning objectives were met. See Table 2 in Section 3.3.4 for a mapping of the discussion questions to the learning objectives.

Table 1

List of Learning Objectives for the Course Module

	Learning Objective
1	Explain what an application logic flaw is.
2	Explain why it is difficult for an automated tool to find logic flaws.
3	Identify potential areas for logic flaws in HTTP interactions.
4	Critique back-end code for HTTP interactions, which has logic flaws.
5	Fix logic flaws in back-end for HTTP interactions to secure it.

3.2 Rationale

The use of technology is widespread and increasing rapidly. The software for these technologies needs to be protected from the consumer level up to critical infrastructures. In 2013, Information Assurance and Security was added as a new Knowledge Area in the ACM/IEEE Computer Science Curricula 2013 [22]. This was in recognition of the importance of software security, and that security concepts need to be taught as part of general undergraduate computer science curriculum.

Existing resources on logic flaws provide very specific categories to test for, but do not show how to handle logic that is custom to an application. The course module was designed to teach both the concept of logic flaws, and give students the skills to identify logic flaws in existing software.

Logic flaws are a complex topic, and not something many students have been exposed to before. The case study provided by Wang et al. (Section 2.4) is difficult to complete without extensive work to understand the material. It is not easily integrated into existing computer

science curricula; the instructor would likely need to spend time reviewing the *How to Shop For Free* article to thoroughly understand it, and then complete the case study themselves since only one solution was provided. The article did not cover prevention of logic flaws during software development, or other resources on manual testing methods like those covered in Section 2.2. As an alternative, this case study is meant to be more easily understood while maintaining the same real-world example, and is part of a complete course module on the topic of logic flaws.

In addition to using the learning objective as a guide for the content of the discussion questions, Blooms Taxonomy was referenced for the types of questions asked. The taxonomy categorizes different levels of learning based on cognitive complexity, and can be used to guide the development of course materials with the goal of increasing student's comprehension and problem solving skills [65]. The discussion questions focused on the cognitive levels of Application, Analysis, and Synthesis

3.3 Course Module Materials

This section covers the introduction, reading quiz, animation, discussion questions, virtual machine for running an automated testing tool, and instructor materials developed for this course module.

3.3.1 Introduction. The Introduction to Logic Flaws includes material previously discussed in Chapters 1 and 2 of this thesis in more detail. Students were also instructed to read the research article the case study is based on, *How to Shop for Free Online: Security Analysis of Cashier-as-a-Service Based Web Stores* [20]. The introduction provides information directly addressing the first two learning objectives in Table 1. Application logic flaws are defined, and where logic flaws come from is covered. Since logic flaws require skill and knowledge to find, the importance of finding and fixing these kinds of vulnerabilities is stressed.

A Section on testing explains how logic flaw security testing can be performed alongside traditional functional testing. Specific testing recommendations from *The Web Application Hacker's Handbook* [5] and OWASP [12] are listed and reviewed, followed by a detailed description of the manual code review method used in the case. Current automated prototypes are reviewed, including the automated tool used by Wang et al., to show the use and limitations of automated testing for logic flaws.

While not covered in the discussion questions, there are resources that provide recommended practices to reduce logic flaws during software development. Security should be addressed throughout the entire software development cycle, and is more effective when intentionally built-in to the software [2, 66, 67]. These recommendations are compiled and organized by the category of logic flaw.

Finally, the case study is described in detail. It explains the exploit in the Bigcommerce and PayPal Express case, and how Wang et al. used the manual code review method to find the exploit, then verified it with an automated tool. The remaining materials, such as the animation and discussion question, are also described.

3.3.2 Reading Quiz. A quiz with ten questions based on the Introduction to Logic Flaws was developed to verify if students actually read the introduction. A variety of question formats are used, such as matching, true/false, multiple-choice, and multi-answer. They can be administered online using an online learning environment such as Blackboard and graded automatically. They were not designed to be difficult, but were used to ensure students read the introduction before viewing the animation and answering the discussion questions.

3.3.3 Case Study Animation. The animation was developed in PowerPoint, and covers the case study. Before the animation, there are several slides reviewing the logic flaw, and how it

was found using the manual code review method. The testing method is summarized with a list of steps on how to manipulate the API arguments and application workflow. The notations used in the animation are explained, such as the difference between an unsigned parameter, which is transmitted in cleartext, and a parameter with a digital signature. Different text colors are used for the three parties involved in the transaction (Attacker, Merchant, and PayPal Express). The three colors are used to signify parameters and back-end code used by each of the three parties. There is a list of the API arguments and public API methods for the Merchant and PayPal Express, which are visible or accessible to the Attacker.

Before starting the animation, a sequence diagram shows the overall HTTP interactions performed when completing a normal checkout transaction (Figure 1). The animation then reviews two manipulated transactions that lead to an exploitable logic flaw. Each step of the checkout has two stages in the animation. First, it animates the HTTP interactions in a sequence diagram.

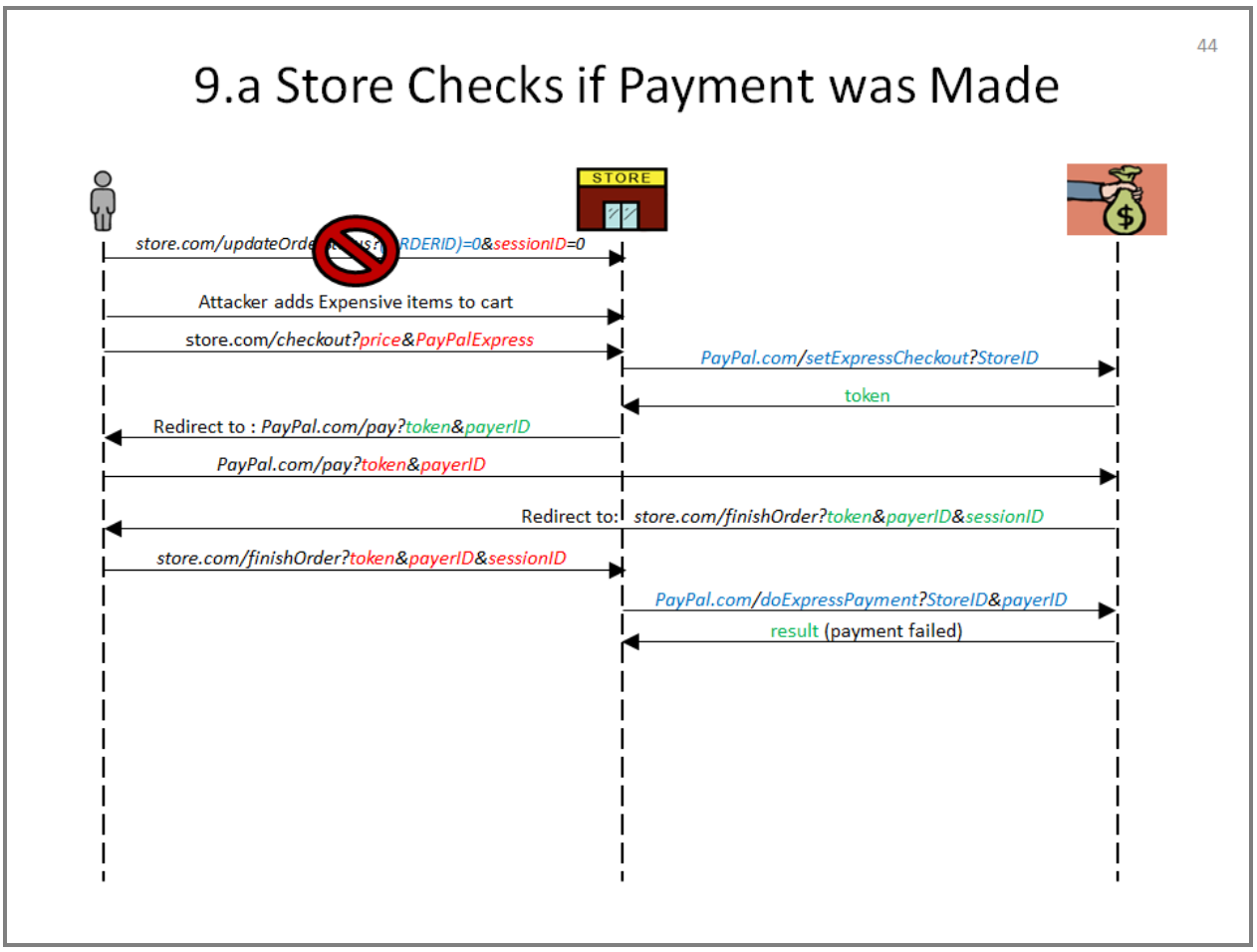

Figure 1 Sequence Diagram with HTTP interactions

All parameters are color-coded by the last party who could modify them. For example, unsigned parameters sent out by the Merchant may be modified by the Attacker. Then, it traces through the back-end code representing that step. As the code is displayed, a table maintaining the current state of the variable values is updated (Figure 2). The table is also color coded to show which variables are used by the Attacker, Merchant, and PayPal Express.

The back-end code used in the animation was modified from code provided in the case study accompanying How to Shop for Free Online, which actually covered four cases. Unneeded code was removed for clarification, such as if-statements based on which third-party cashier was being used. Since only PayPal Express was used for this course module, any references to

Amazon SimplePay, Google Checkout, and PayPal Standard in the original case study were removed. The API method names used in the course module were modified to match the method names used in the *How to Shop for Free Online* article.

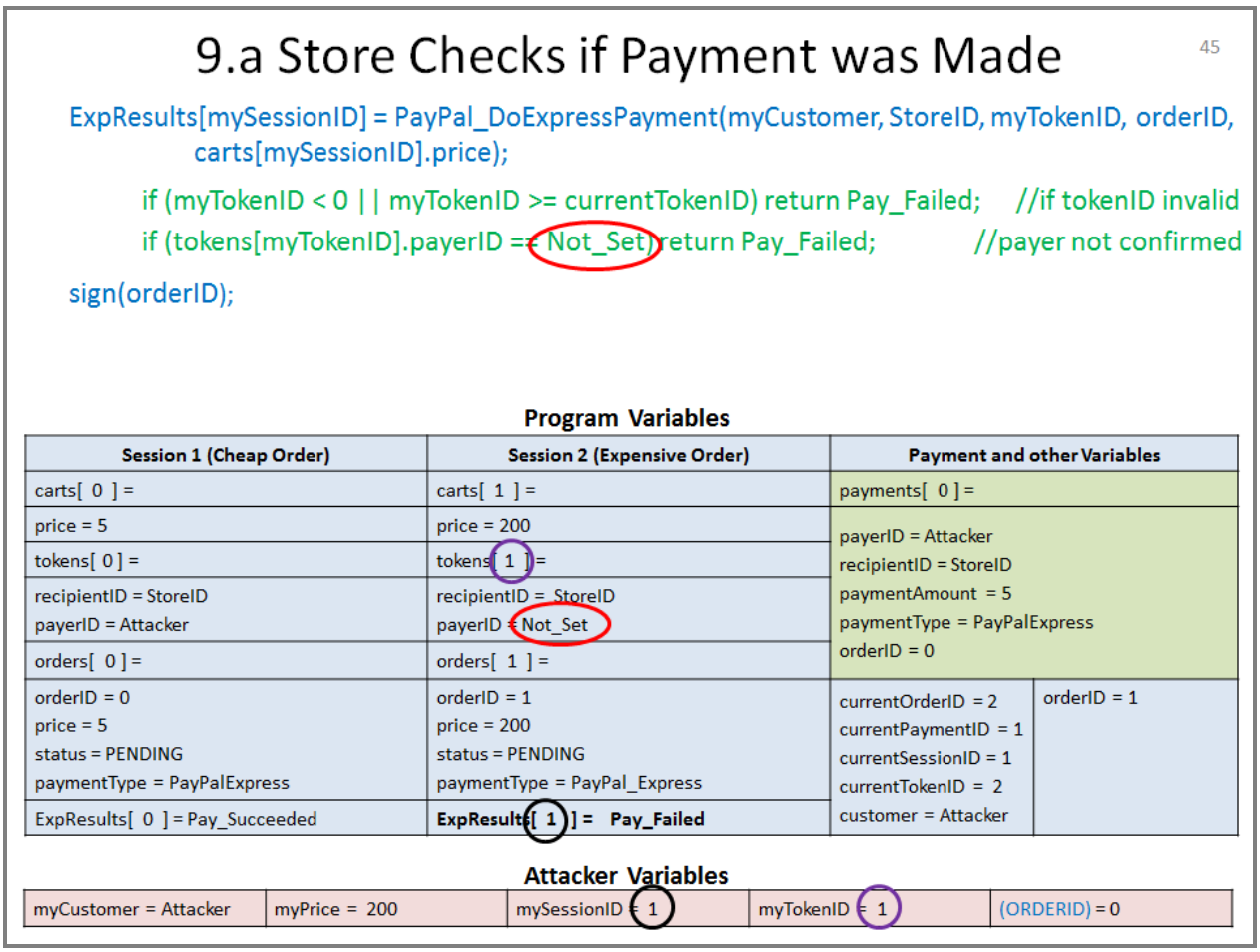

Figure 2 Back-end code accompanying sequence diagram in Figure 1

Any variables that were accessible or set by the attacker were changed to the format: myVariable. This helps show how the attacker-accessible variables are used by the Merchant and PayPal Express throughout the entire checkout transaction.

After the animation Section is complete, there are slides that explain what steps the Merchant and PayPal Express took to ensure the transaction was secure and correct, such as

verifying that payment was attempted and did not fail. Then the logic flaw is described in detail, explaining which steps led to the exploit.

3.3.4 Discussion Questions. There are eight discussion questions, most of which are based on the learning objectives. These questions were designed to be difficult, and intended to be given to students in groups of 2-3. Not all questions need to be given to students; a subset of the questions can be chosen. Some questions could be used in a quiz or test. See Table 2 for a mapping of discussion questions to Blooms Taxonomy cognitive levels, and to the learning objectives.

Table 2

Mapping of Discussion Questions to Learning Objectives and Blooms Taxonomy

Learning Objective ID	Question ID	Case Study Discussion Questions	Blooms Taxonomy Cognitive Level
1, 2	1	What is a logic flaw, and why are they difficult to identify?	Level 1 - Knowledge Level 2 - Comprehension
	2	Why is it NOT a security vulnerability that the token, payerID, sessionID, and PayPalExpress URL parameters from the Bigcommerce and PayPal Express case are unsigned?	Level 4 - Analysis

Table 2

Cont.

3	3	Read through the following URLs representing HTTP interactions between three parties. Suggest potential security flaws for each of the arguments.	Level 4 - Analysis
4	4	Read through the following code and critique it.	Level 6 - Evaluation
5	5	Fix the following code to better secure it.	Level 5 - Synthesis
	6	Propose methods to test for logic flaws (does not have to be automated)	Level 5 - Synthesis
	7	Given the following code and variable table, trace through the variables at each stage.	Level 3 - Application
	8	The following URLs/code sequence is out of order. What is the correct sequence?	Level 3 - Application

The discussion questions are listed with more detail in Appendix C.

Questions 1 and 2 ask students about what logic flaws are, and about the case study from the animation. Questions 3-5 are part of a set, and use the same example code. This code represents the case Bigcommerce and Google Checkout from *How to Shop for Free Online*, and modified for clarification, in a similar manner as the PayPal Express case. The questions are about identifying potential security flaws in API arguments using the HTTP interactions, identifying potential security flaws in the back-end code for the case, and then fixing any

discovered flaws. Students are directed to use the manual code review method outlined in the animation.

Question 6 asks students to propose testing methods for finding logic flaws. The last two questions, 7 and 8, use the case Bigcommerce and PayPal Standard. This code was also modified from the original case study for clarity. Only one question from this set should be chosen to give to the students. The first question asks students to trace through the given back-end code and fill out a table representing the current state of the variable values, similar to the animation. The second question provides the code with the trace table already completed; however, the steps of the checkout transaction are out of order. Students must find the correct order of API method calls.

3.3.5 An Automated Testing Tool - Corral. VMware was used to set up a Virtual Machine running Windows 7 with the prerequisites needed to run Corral, the automated tool used by Wang et al. to confirm the logic flaws discovered using manual methods. This machine was distributed to students with code from the original case study designed for use with Corral. Students ran the tool, and submitted their review on its uses and limitations. Due to licensing of the operating system, this Virtual Machine will not be made available to other universities along with the other course module materials. However, instructions for setting up a Virtual Machine for installing and running Corral will be provided, as the prerequisites for Corral and Corral itself are either open-source or available for free.

3.3.6 Instructor Materials. Instructors are provided with the answers to the reading quiz, and a rubric for grading the discussion questions. Since different subsets of the discussion questions may be chosen, and different point values may be given to each question, the rubric provides a percentage of points to assign for each answer instead of fixed point values. For

example, the question asking students to suggest potential security flaws for each argument in a set of API methods has 6 total parameters. If 5 or more parameters are answered correctly, 100% of the points are given for this question. If only 4 are correct, 75% of the points are given.

Some of the questions were designed to be open-ended and use higher levels of thinking. Potential answers are given in the rubric, but they are not the only acceptable answers to these questions.

3.4 Course Module Evaluation

Prior to starting the course module, students were given a pre-survey asking them to rate their level of knowledge of application logic flaws, based on the learning objectives, on a 5-point Likert scale. A post-survey with the same questions from the pre-survey was given once the course module was completed. It also asked students to review the usefulness of the module on a 5-point Likert scale, and what they learned or problems that were encountered as open answer questions. These surveys were approved by the Institutional Review Board (IRB) at North Carolina A&T State University, and are listed in Appendix B.

The surveys and reading quizzes were completed individually, and the discussion questions were completed in groups of 3. As this course module was taught in an online course, students were instructed to record their discussion in a Blackboard online forum. One forum was created for each group. The forums allowed the discussions to be reviewed for how active or lively they were.

CHAPTER 4

Results

This course module was taught in COMP 727 Secure Software Engineering at North Carolina A&T State University in the Spring 2015 semester. There were 21 students registered for the course. Students were given approximately three weeks, including spring break, to complete the course module.

4.1 Survey Results

A voluntary pre-survey and post-survey, approved by the IRB and North Carolina A&T State University, was given to students in COMP 727 Secure Software Engineering via Blackboard. The surveys asked students to rate their level of knowledge or skills for each of the learning objectives described in Table 1. The fourth and fifth learning objectives were simplified in the surveys to "Critique code with logic flaws" and "Fix code with logic flaws to secure it". The surveys were also anonymous, but students were asked to provide a 4-digit ID number so paired statistical analysis could be done. Participants in the survey were graduate-level computer science students. More detailed demographics were not collected. Aggregate survey results are listed in Appendix D,

Out of 21 students registered for the course, three submitted blank or partially blank pre-surveys. For the post-survey, 17 surveys were completed, and three were blank. As an unequal number of surveys were completed for the pre- and post-surveys, and in the post-survey several students declined to respond with the four-digit number being used to pair responses, a paired statistical analysis could not be performed. Instead, a Mann-Whitney U test was more appropriate, with the before and after data treated as independent samples. It is a nonparametric test of the null hypothesis that two populations are equal.

The null hypothesis H_0 is that the two populations are equal, and the research hypothesis H_1 is that the two populations are not equal. With $N_1 = 18$, $N_2 = 14$, and $\alpha = 0.05$, the calculated critical value for this test is 74 [68]. Thus, H_0 is rejected if $U \leq 74$.

Table 3

Mann-Whitney U test results of students' ranking of their knowledge/skills on logic flaws on a 5-point Likert scale

Learning Objective:	**1**	**2**	**3**	**4**	**5**
U	49.5	72.5	41	57	67
Asymptotic 2-tailed *p*-value	.002	.032	.001	.004	.013

The results in Table 3 show that students felt they improved their knowledge or skills for most of the learning objectives described in Table 1. All of the *p*-values indicate these results are statistically significant, and H_0 is rejected for each learning objective. As this was calculated using a relatively small sample size, the results of this Mann-Whitney U test may not be representative of the population of university-level computer science students.

The post-survey included additional questions asking students to rate their interest in the topic of logic flaws and to review the course module.

Table 4

Results of how much the students agree or disagree with the statements on the post-survey

Question	Response
The case study was useful to help you understand the material	29% Strongly Agree 50% Agree

Table 4

Cont.

You were motivated to learn about application logic flaws	29% Strongly Agree 29% Agree
You enjoyed the case study	29% Strongly Agree 36% Agree
The learning objectives were met	7% Strongly Agree 71% Agree
The case study helped you to develop problem solving and critical thinking skills, and ability to think independently	21% Strongly Agree 43% Agree

Overall, these responses show students enjoyed the case study, and felt it was useful. Most importantly, 78% of students felt the learning objectives were met and 63% of the students felt they improved their problem solving and critical thinking skills.

The post-survey also asked students several open-ended questions: what was the most important thing they learned, if any problem were encountered while completing the case study, and if they had any additional comments. The most common things they learned were: 1) never trusting input from the client side of a web application, and 2) how seemingly simple vulnerabilities can lead to serious logic flaws. A few comments reiterated that students enjoyed the analytical thinking. Problems were encountered with 1) running Corral on the virtual machine, 2) understanding the article by Wang et al., and 3) with tracing the state of variables through each step of the transaction when doing the manual code review. Notable comments

were: “It was an interesting concept to learn.” and “The case study was really good. It helped me a lot in enriching knowledge on application logic flaws.”

4.2 Graded Reading Quiz

The average score on the quiz about the reading was 89.80. Quiz grades are listed in Appendix D. Out of ten questions, four were missed by at least 20% of the class.

- One question was matching the terms bug, flaw, logic flaw, and vulnerability to their definitions. Three students mixed up flaws and logic flaws, and two mixed up flaws, logic flaws, and vulnerabilities. The definition of a software vulnerability can be added to the introduction on logic flaws.
- One question was multiple-choice, asking about what can cause flaws during development. Four students answered with the example of a bug or a flaw, instead of elements of the software development lifecycle that can affect the quality of the software. In the introduction, the section on what causes flaws can be expanded or clarified.
- One question was on the possible number of flaws in a software application, given the number of lines of code, and the number of vulnerabilities per thousand lines of code. It was emphasized that the question asked for the number of flaws only. Five students answered with the total number of vulnerabilities, which included both bugs and flaws. These students also had incorrect answers to the term matching question. There is an example in the introduction that estimates the number of flaws for an application; this can be expanded to also estimate the number of bugs, to help make a distinction between the two.
- For one question, half the class’ answers were partially incorrectly. The question was on methods that can be used to find logic flaws; multiple answers could be selected. Nine

students selected the three correct answers, but also one incorrect answer, indicating either the question or the introduction was not clear enough on this subject. The introduction can be clarified to say that logic flaw testing can be done alongside functional testing, but functional testing is not used to specifically find logic flaws. The question can also be modified to ask which testing methods are used primarily for finding logic flaws.

4.3 Discussion Questions

While the reading quiz and surveys were individual, the discussion questions were assigned to students in groups of three, for a total of seven groups. Out of the eight available questions for the case study covered in Section 3.3.4, five were chosen: questions 2-5 and 8 from Table 2. The last question for the assignment was to run Corral on a virtual machine using the C header file and symbolic driver program mentioned in Section 2.3, and describe their experience using an automated prototype to test for logic flaws. Question 1 from Table 2 was given on a mid-term exam after the course module was completed.

Students' grades for the course module were based on several factors. Their individual participation and the quality of their posts on the online Blackboard forum was worth 25 points. The quality of the discussion for the group as a whole, based on the forum posts, worth of 20 points. Finally, the answers to the discussion questions themselves in the final group report were worth 55 points, for a total of 100 points for the course module.

A limitation of the results presented for this section is that members from at least three groups participated in one large discussion. The reports for these groups were based on this collaboration, and therefore have similar content. This means there are essentially four unique sets of responses that can be evaluated instead of seven.

4.3.1 Quality of Discussion Overall the discussion on the online Blackboard forums was high-quality and indicated that most of the groups had very active discussions. Some groups met in person and recorded their conversations as best as possible, while other groups communicated almost entirely through the Blackboard forums. The posts show that students were in fact able to perform the high level of analysis, synthesis, and evaluation that was asked of them. Table 5 shows excerpts from student online discussion of one group.

Table 5

Excerpts from student online discussions

"You are right about number 1. I was wrong when I said that modifying myPrice would have any kind of effect. Since it's not being used in the code sample we have then there is not a vulnerability present. But for #3…I think that it is still vulnerable to a replay attack. What if we initiated an order with a cheap piece of merchandise in order to get a signed SESSIONID, CART, and IPNHANDLER"
"I think you are right. We may have been overthinking this stage of the order processing. If the cahsier sends over the price information for verification then that would be sufficient as long as the merchant can always verify the origin of the message as coming from a cashier and not from an attacker."

The online discussions will be also be used as part of a research project at A&T State University on collaborative learning.

4.3.2 Graded Discussion Questions The rubric described in Section 3.3.6 was used to grade each of the discussion questions presented in the reports submitted by each group. The five main discussions questions covered in Section 4.3 were out of 10 points each, and the question

on students experience running Corral was 5 points. Grades ranged between 42.5 and 55, out of 55 points, as shown in the box-and-whisker plot in Figure 3. Discussion question grades are listed in Appendix D.

The question students struggled with the most was question 3 (Table 2). Instructions mentioned that this was related to the BigCommerce and Google Checkout case in the research article by Wang et al. Instead of providing their own responses based on the HTTP arguments, students summarized each step of the code, referencing the article. The rubric was based on general potential vulnerabilities, and not on the source code behind the HTTP request. For example, unsigned arguments are sent in cleartext and could be modified, and the source code may not include verifications to check the validity of the arguments' value.

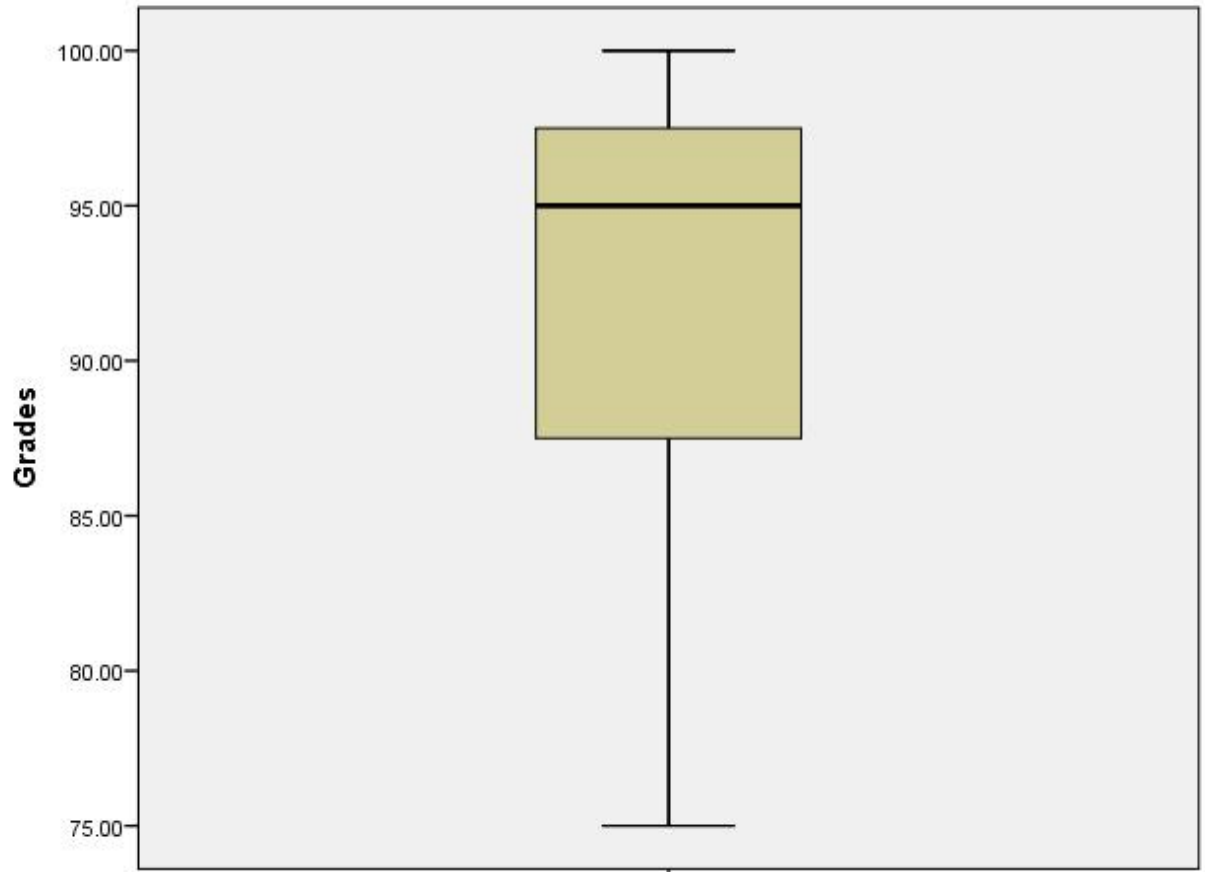

Figure 3 Box and Whisker graph of graded discussion questions

Question 4 (Table 2) seemed to be the question that students found most difficult. The three groups who were not involved in the multi-group collaboration did an excellent job on this question. They identified both vulnerabilities that led to a logic flaw and security measures present in the provided source code. The collaborated answer relied more on the article by Wang et al. than on the students' own analysis, and did not mention the existing security measures. Some of their identified vulnerabilities were also incorrect. Question 5 (Table 2) used analysis from the previous question to fix the logic flaw; incorrect analysis led to some incorrect mitigation of vulnerabilties in the code. Most of the groups did attempt the correct method for fixing the logic flaw present in this code, although it was not always properly implemented.

For question 1 (Table 2) on the mid-term exam, the average grade was an 8.3 out of 10 points, and 12 students received the full 10 points. Students were able to describe what logic flaws were, and provided examples. They also understood why logic flaws are difficult to identify, for both manual and automated testing methods.

CHAPTER 5

Discussion and Future Research

Based on the overall results, this course module was successful at teaching graduate-level computer science students about application logic flaws. It adds to the number of case studies available for computer science education, specifically supporting secure software engineering or software security testing courses. It should help bridge the gap that exits between academia and industry best practices for software engineering.

Future work can be done to improve the course module as a whole, and more case studies on logic flaws can be developed to supplement the existing case. The animation relies on correct alignment of images and text, but may be formatted incorrectly when opened in applications other than PowerPoint 2010 on Windows computer. By switching to a cross-platform medium such as HTML 5, the animation formatting will be more consistent when viewed on different devices. As mentioned in Section 4.2, one of the reading quiz questions was partially incorrect by half of the class. Both the introduction document and the question can be clarified.

The grading rubric for the discussion questions (Section 3.3.6), does not contain all possible answers, since some of the questions are open-ended. Student responses to these questions included quality answers not currently listed in the rubric; these can be added to expand the set of possible answers.

For the case study, students recommended that the animation include a description of the source code. Each step showed the HTTP interactions in a sequence diagram, followed by the back-end code. A summary can be added to each step, explaining what is actually happening. For example: The attacker added one cheap item to their cart, and started the checkout process. Some of the terminology used in the article by Wang et al. and the case study was new to the students,

which led to difficulty understanding the details of application logic flaws. Both the introduction to logic flaws and the animation can be updated to include a list explaining the terms being used. Code used in the animation was modified to clarify which arguments were accessible by the attacker, but confused the students if they tried comparing the case study source code to the examples given in the article by Wang et al. Before the animation starts, the modifications can be listed and explained.

Students said tracing the state of variables through each step of the transaction when doing the manual code review was difficult. Either a table similar to the one used in the animation can be provided as part of that discussion question, or the introduction and the animation can be updated to explain the steps for creating their own table. They also had a difficult time coming up with ways to mitigate the logic flaws in the code. A section in the introduction and the animation can be added to address this explicitly. For example, in the case study the sessionID could be bound to the orderID, and the digitally signed argument containing the shopping cart information should not be created if payment to PayPal Express failed, or at least not made available to the user. Finally, the instructions for running Corral on the virtual machine can use some clarification, as well as an explanation of what the scan results mean.

As this course module has only been taught and evaluated once, with a relatively small sample size, future work would include using the module in other universities or again at NC A&T State University. It can be taught as part of a software engineering or software security testing course, as discussed in Section 1.3. Several professors from other universities have already expressed their interest in using this course module as part of their curriculum.

References

[1] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*. Stoughton, MA: Addison-Wesley, 2004.

[2] G. McGraw, *Software Security: Building Security In*. Crawfordsville, IN: Addison-Wesley, 2006.

[3] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, 2nd ed. Redmond, WA: Microsoft Press, 2004.

[4] D. McCandless et al., Codebases: Millions of lines of code, Information is Beautiful, 26 Nov. 2014. [Online]. Available: http://www.informationisbeautiful .net/ visualizations/million-lines-of-code/ [Accessed: 19 Mar. 2015].

[5] D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*, 2nd ed. Indianapolis, IN: John Wiley, 2011.

[6] J. Grossman, "Seven Business Logic Flaws That Put Your Website At Risk," WhiteHat Security, Santa Clara, CA, White Paper, 2007. [Online]. Available: https://www.whitehatsec.com/assets/ WP_bizlogic092407.pdf [Accessed: 19 Mar. 2015].

[7] Net Market Share, *Market Share Reports: Desktop Operating System Market Share*, Net Applications, Jan. 2015. [Online]. Available: http://www.netmarketshare.com/ [Accessed: 7 Feb. 2015].

[8] R. Los and P. Jagdale. "Defying Logic: Theory, Design, and Implementation of Complex Systems for Testing Application Logic," in *Black Hat Europe 2011 – Briefings*. [Online]. Available: http://www.slideshare.net/RafalLos/defying-logic-business-logic-testing-with-automation [Accessed: 19 Mar. 2015].

[9] Hewlett-Packard Development Company, *FORTIFY STATIC CODE ANALYZER*, Hewlett-Packard Development Company, 2015. [Online]. Available: http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/index.html? [Accessed: 14 Mar. 2015].

[10] International Business Machines Corporation (IBM), *IBM Security AppScan Source*, IBM. [Online]. Available: http://www-03.ibm.com/software/products/en/appscan-source [Accessed: 14 Mar. 2015].

[11] CWE Content Team, *CWE-603: Use of Client-Side Authentication*, The MITRE Corporation, Jul. 30 2014. [Online]. Available: http://cwe.mitre.org/data/definitions/603.html [Accessed: 14 Mar. 2015].

[12] Open Web Application Security Project, *Testing for Business Logic*, Open Web Application Security Project, Aug. 5 2014. [Online]. Available: https://www.owasp.org/index.php/Testing_for_business_logic [Accessed: 19 Mar. 2015].

[13] W. Perry, *Effective Methods for Software Testing*, 3rd ed. Indianapolis, IN: John Wiley, 2006.

[14] Ponemon Institute, "2012 Web Session Intelligence & Security Report: Business Logic Abuse Edition," Ponemon Institute, Traverse City, MI, Research Report, 2012. [Online]. Available: http://www.emc.com/collateral/rsa/silvertail/rsa-silver-tail-ponemon-ar.pdf [Accessed: 19 Mar. 2015].

[15] F. Shull et al.. "What We Have Learned About Fighting Defects," in *Proc. of the 8th Int. Symp. on Software Metrics (METRICS '02)*. [Online]. Available: http://www.cs.umd.edu/~mvz/ pub/eworkshop02.pdf [Accessed: 19 Mar. 2015].

[16] C. Jones and O. Bonsignour, *The Economics of Software Quality*. Westford, MA: Pearson, 2012.

[17] M. Howard and D. LeBlanc, *Writing Secure Code*, 2nd ed. Redmond, WA: Microsoft Press, 2003.

[18] C. Davis and E. Wilcock, *Teaching Materials Using Case Studies*. Liverpool, UK: The UK Centre for Materials Education, 2003. [Online] Available: http://www.materials.ac.uk/guides/1-casestudies.pdf [Accessed: 19 Mar. 2015].

[19] I. Baumgartner and V. Shankararaman, "Case studies in computing education: presentation, evaluation and assessment of four case study-based course design and delivery models," in *Frontiers in Education Conf. (FIE)*, IEEE, 22-25 Oct 2014, pp. 1-8. [Online]. Available: IEEE Xplore, doi: 10.1109/FIE.2014.7044194 [Accessed: 19 Mar. 2015].

[20] R. Wang et al., "How to Shop for Free Online: Security Analysis of Cashier-as-a-Service Based Web Stores," in *2011 IEEE Symp. on Security and Privacy (SP)*, IEEE, pp. 465-480. [Online]. Available: IEEE Xplore, doi: 10.1109/SP.2011.26 [Accessed: 19 Mar. 2015].

[21] M. Forehand, *Bloom's Taxonomy: Original and Revised*, Emerging Perspectives on Learning, Teaching, and Technology, Sept. 8 2014. [Online]. Available: http://epltt.coe.uga.edu/index.php?title=Bloom%27s_Taxonomy [Accessed: 19 Mar. 2015].

[22] Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society, *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*, 2013. [Online]. Available: ACM, doi: 10.1145/2534860 [Accessed: 19 Mar. 2015].

[23] N. Mead et al.., "Software Assurance Curriculum Project Volume II: Undergraduate Course Outlines," Software Engineering Institute, Tech. Rep. CMU/SEI-2010-TR-019, Pittsburgh, PA, Aug 2010. [Online]. Available:

http://resources.sei.cmu.edu/asset_files/TechnicalReport/ 2010_005_001_15251.pdf [Accessed: 19 Mar. 2015].

[24] N. Mead et al.., "Software Assurance Curriculum Project Volume I: Master of Software Assurance Reference Curriculum," Software Engineering Institute, Tech. Rep. CMU/SEI-2010-TR-005, Pittsburgh, PA, Aug 2010. [Online]. Available: http://resources.sei.cmu.edu/asset_files/TechnicalReport/ 2010_005_001_78257.pdf [Accessed: 19 Mar. 2015].

[25] A. Aditomo et al., "Inquiry-based learning in higher education: principal forms, educational objectives, and disciplinary variations," *Studies in Higher Education*, vol. 38, no. 9, 2013, pp. 1239-1258. [Online]. Available: Taylor Francis Online, doi: 10.1080/03075079.2011.616584 [Accessed: 15 Mar. 2015].

[26] A. Cullen., "Using the Case Method to Introduce Information Skill Development in the MBA Curriculum," *Journal of Business & Finance Librarianship*, vol. 18, no. 3, Jun. 2013, pp. 208-232. [Online]. Available: Taylor Francis Online, doi: 10.1080/08963568.2013.795740 [Accessed: 15 Mar. 2015].

[27] R. Stark et al.., "Case-based learning with worked examples in complex domains: Two experimental studies in undergraduate medical education," *Learning and Instruction*, vol. 21, no. 1, Feb 2011, pp. 22-33. [Online]. Available: Science Direct, doi: 10.1016/j.learninstruc.2009.10.001 [Accessed: 15 Mar. 2015].

[28] S. Lee et al., "A review of case-based learning practices in an online MBA program: A program-level case study," *Educational Technology & Society*, vol. 12, no. 3, 2009, pp. 178-190. [Online]. Available: http://www.ifets.info/journals/12_3/16.pdf [Accessed: 15 Mar. 2015].

[29] K. Reid, "Building a Community of Scholars: One University's Comparison of "Typical" vs. Open Ended Ethics Case Studies in First-Year Engineering," *Journal of STEM Education: Innovations and Research*, vol. 13, no. 4, Sept 2012, pp. 18-23. [Online]. Available: http://www.jstem.org/index.php?journal=JSTEM&page=article&op=view&path[]=1663 [Accessed: 15 Mar. 2015].

[30] A. Yadav et al., "Lessons Learned: Implementing the Case Teaching Method in a Mechanical Engineering Course," *Journal of Engineering Education*, vol. 99, no. 1, Jan. 2010, pp. 55-69. [Online]. Available: DOI: 10.1002/j.2168-9830.2010.tb01042.x [Accessed: 15 Mar. 2015].

[31] Q. Le, "Implementation of Case Studies in an Introduction to Engineering Course for "LITEE National Dissemination Grant Competition"," *Journal of STEM Education: Innovations and Research*, vol. 13, no. 4, Sep. 2012, pp. 12-17. [Online]. Available: http://www.jstem.org/index.php?journal=JSTEM&page=article&op=view&path[]=1558 [Accessed: 15 Mar. 2015].

[32] A. Clayson, "Effectiveness of LITEE Case Studies in Engineering Education: A Perspective from Genre Studies," *Journal of STEM Education: Innovations and Research*, vol. 12, no. 7, 2011, pp. 15-31. [Online]. Available: http://www.jstem.org/index.php?journal=JSTEM&page=article&op=view&path[]=1508 [Accessed: 15 Mar. 2015].

[33] L. Cassel et al., *Computer Science Curriculum 2008: An Interim Revision of CS 2001*, 2008. [Online]. Available: ACM, http://www.acm.org/education/curricula/ComputerScience2008.pdf [Accessed: 15 Mar. 2015].

[34] X. Yuan et al., "Case Studies for Teaching Physical Security and Security Policy," *in 2010 Information Security Curriculum Development Conf. (InfoSecCD '10)*, ACM, pp. 3-12. [Online]. Available: ACM, doi: 10.1145/1940941.1940947. [Accessed: 15 Mar. 2015].

[35] X. Yuan et al., "Teaching Security Management with Case Studies: Experiences and Evaluation," *Journal on Education, Informatics and Cybernetics (JEIC)*, vol. 2, no. 2, 2010, pp. 25-30. [Online]. Available: http://www.iiis.org/CDs2010/CD2010SCI/EISTA_2010/PapersPdf/EA605YI.pdf [Accessed: 19 Mar. 2015].

[36] J.E. Thistlethwaite et al., "The effectiveness of case-based learning in health professional education. A BEME systematic review: BEME Guide No. 23," *Medical Teacher*, vol. 34, no. 6, 2012, pp. e421-e444. [Online]. Available: doi:10.3109/0142159X.2012.680939 [Accessed: 19 Mar. 2015].

[37] A. Çam and Ö Geban, "Effectiveness of Case-Based Learning Instruction on Epistemological Beliefs and Attitudes Toward Chemistry," *Journal of Science Education and Technology*, vol. 20, no. 1, Feb. 2011, pp. 26-32. [Online]. Available: Springer, doi: 0.1007/s10956-010-9231-x [Accessed: 19 Mar. 2015].

[38] E. Pyatt, *Using Cases in Teaching*, Penn State University, May 11 2006. [Online]. Available: http://archive.tlt.psu.edu/suggestions/cases/ [Accessed: 15 Mar. 2015].

[39] M. Daun et al., "Industrial case studies in graduate requirements engineering courses: The impact on student motivation," in *2014 IEEE 27th Conf. on Software Engineering Education and Training (CSEE&T)*, IEEE, pp. 3-12. [Online]. Available: IEEE Xplore, doi: 10.1109/CSEET.2014.6816775 [Accessed: 15 Mar. 2015].

[40] H. Beckman et al., "Collaborations: closing the industry-academia gap," *IEEE Software*, vol. 14, no. 6, Dec. 1997, pp. 49-57. [Online]. Available: IEEE Xplore, doi: 10.1109/52.636668 [Accessed: 15 Mar. 2015].

[41] M. Shaw (Ed.), "Software Engineering for the 21st Century: A basis for rethinking the curriculum," Tech. Rep. CMU-ISRI-05-108, Carnegie Mellon University, Institute for Software Research International, Pittsburgh, PA, 2005. [Online]. Available: https://www.cs.cmu.edu/~Compose/SEprinciples-pub-rev2.pdf [Accessed: 19 Mar. 2015].

[42] A. J. LaSalle., "An inverted computing curriculum: preparing graduates to build quality systems," in *27th Annual Frontiers in Education Conf.*, IEEE, 5-8 Nov 2014, pp. 280-284. [Online]. Available: IEEE Xplore, doi: 10.1109/FIE.1997.644857 [Accessed: 15 Mar. 2015].

[43] V. Varma and K. Garg, "Case studies: the potential teaching instruments for software engineering education," in *5th Int. Conf. on Quality Software (QSIC 2005)*, IEEE, pp. 279-284. [Online]. Available: IEEE Xplore, doi: 10.1109/QSIC.2005.18 [Accessed: 19 Mar. 2015].

[44] C. Wohlin and B. Regnell, "Achieving industrial relevance in software engineering education," in *12th Conf. on Software Engineering Education and Training (CSEE&T)*, IEEE, 22-24 Mar 1999, pp. 16-25. [Online]. Available: IEEE Xplore, doi: 10.1109/CSEE.1999.755175 [Accessed: 19 Mar. 2015].

[45] W. He et al., "Supporting Case-based Learning in Information Security with Web-based Technology," *Journal of Information Systems Education*, vol. 24, no. 1, 2013, pp. 31-40. [Online]. Available: http://jise.org/Volume24/24-1/PDF/Vol24-1pg31.pdf [Accessed: 19 Mar. 2015].

[46] L. Yang et al., "Develop Case Studies to Teach Cryptography in A Collaborative Environment," in *2011 Int. Conf. on Frontiers in Education: Computer Science and Computing Engineering*, 18-21 Jul. 2011. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.217.7220&rep=rep1&type=pdf [Accessed: 19 Mar. 2015].

[47] Common Weakness Enumeration, *CWE-840: Business Logic Errors*, The MITRE Corporation, Jul. 3 2014. [Online]. Available: http://cwe.mitre.org/data/definitions/840.html [Accessed: 15 Mar. 2015].

[48] The MITRE Corporation, *Common Attack Pattern Enumeration and Classification*, The MITRE Corporation, Nov. 7 2014. [Online]. Available: http://capec.mitre.org/ [Accessed: 15 Mar. 2015].

[49] Open Web Application Security Project, *Business Logic Security Cheat Sheet*, Open Web Application Security Project, Jun. 5 2014. [Online]. https://www.owasp.org/index.php/Business_Logic_Security_Cheat_Sheet [Accessed: 15 Mar. 2015].

[50] J. Jurcenoks, "OWASP to WASC to CWE Mapping: Correlating Different Industry Taxonomy," Critical Watch, Dallas, TX, White Paper, Jun. 2013. [Online]. Available: http://www.criticalwatch.com/assets/c-Owasp-to-Wasc-to-CWE-Mapping-Tech-Paper-0710131.pdf [Accessed: 19 Mar. 2015].

[51] Open Web Application Security Project, *Top 10 2013-Top 10*, Open Web Application Security Project, 26 Aug. 2014. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-Top_10/ [Accessed: 19 Mar. 2015].

[52] B. Martin et al., *2011 CWE/SANS Top 25 Most Dangerous Software Errors*, The MITRE Corporation, 6 Jul. 2011. [Online]. Available: http://cwe.mitre.org/top25/ [Accessed: 19 Mar. 2015].

[53] NT OBJECTives, "Top 10 Business Logic Attack Vectors: Attacking and Exploiting Business Application Assets and Flaws – Vulnerability Detection to Fix," NT OBJECTives, Irvine, CA, White Paper, 3 May 2012. [Online]. Available: http://www.ntobjectives.com/research/web-application-security-white-papers/business-logic-attack-vectors-white-paper/ pdf [Accessed: 19 Mar. 2015].

[54] MSDN Library, *Testing Overview*, Microsoft. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa292191%28v=vs.71%29.aspx [Accessed: 1 Jan. 2015].

[55] C. Wysopal et al., *The Art of Software Security Testing*. Crawfordsville, IN: Addison-Wesley, 2007.

[56] G. Pellegrino and D. Balzarotti, "Toward Black-Box Detection of Logic Flaws in Web Applications," in *Network and Distributed System Security Symp. 2014 (NDSS '14)*. [Online]. Available: Internet Society, doi: 10.14722/ndss.2014.23021 [Accessed: 19 Mar. 2015].

[57] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward Automated Detection of Logic Vulnerabilities in Web Applications," in *19th USENIX Conf. on Security (USENIX Security '10)*, ACM. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929820.1929834 [Accessed: 19 Mar. 2015].

[58] SeleniumHQ: Browser Automation, Selenium Project, Dec. 23 2014. [Online]. Available: http://www.seleniumhq.org/ [Accessed: 1 Jan. 2015].

[59] *The Daikon invariant detector*, Massachusetts Institute of Technology and University of Washington, Dec 22 2014. [Online]. Available: http:// plse.cs.washington.edu/daikon/ [Accessed: 15 Mar. 2015].

[60] *Daikon Invariant Detector Use Manual*, Massachusetts Institute of Technology and University of Washington, Mar. 3 2015. [Online]. http://plse.cs.washington.edu/daikon/download/doc/daikon.html#Daikon-runs-slowly [Accessed: 15 Mar. 2015].

[61] A. Das et al., *Q Program Verifier*, Microsoft Research, 2015. [Online]. http://research.microsoft.com/en-us/projects/verifierq/ [Accessed: 15 Mar. 2015].

[62] A. Lal et al., *Corral: Capturing Concurrency*, Microsoft Research, 2015. [Online]. http://research.microsoft.com/en-us/projects/verifierq/corral.aspx [Accessed: 15 Mar. 2015].

[63] L. de Moura et al., *Z3*, Microsoft, 10 Dec. 2014. [Online]. https://z3.codeplex.com/ [Accessed: 15 Mar. 2015].

[64] R. Wang et al., *A Case Study of Cashier-as-a-Service Based Merchant Logic*, Microsoft Research, 2015. [Online]. http://research.microsoft.com/en-us/people/shuochen/caaslogiccasestudy.aspx [Accessed: 15 Mar. 2015].

[65] F. Kugelman, *Bloom's Taxonomy,* bloomstaxonomy.org. [Online]. http://www.bloomstaxonomy.org/Blooms%20Taxonomy%20questions.pdf [Accessed: 15 Mar. 2015].

[66] S. Lipner, "The Trustworthy Computing Security Development Lifecycle," in *Proc. of the 20th Annual Computer Security Applications Conf. (ACSAC '04)*, pp. 2-13. [Online]. Available: ACM, doi: 10.1109/CSAC.2004.41 [Accessed: 19 Mar. 2015].

[67] M. Howard and S. Lipner, *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*, Redmond, WA: Microsoft Press, 2006.

[68] Boston University School of Public Health, *Nonparamnetric Tests*, Boston University School of Public Health, May 13 2013. [Online]. Available: http://sphweb.bumc.bu.edu/otlt/MPH-Modules/BS/BS704_Nonparametric/mobile_pages/BS704_Nonparametric4.html [Accessed: 15 Mar. 2015].

Appendix A

Table A-1 [50]

OWASPs' 2013 Top 10 [51] cross-linked with 2011 CWE/SANS Top 25 Most Dangerous Software Errors [52]

Bug/Flaw Categorization		OWASP Top 10 2013			2011 CWE/SANS Top 25
Bug	A1	Injection	1	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
			2	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
Flaw	A2	Broken Authentication and Session Management	5	CWE-306	Missing Authentication for Critical Function
			7	CWE-798	Use of Hard-coded Credentials
			10	CWE-807	Reliance on Untrusted Inputs in a Security Decision

Table A-1 [50]

Cont.

Flaw	A2	Broken Authentication and Session Management	14	CWE-494	Download of Code Without Integrity Check
			21	CWE-307	Improper Restriction of Excessive Authentication Attempts
Bug	A3	Cross-Site Scripting (XSS)	4	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
Flaw	A4	Insecure Direct Object References	6	CWE-862	Missing Authorization
			9	CWE-434	Unrestricted Upload of File with Dangerous Type
			13	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
			15	CWE-863	Incorrect Authorization

Table A-1 [50]

Cont.

Flaw	A4	Insecure Direct Object References	16	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
Flaw	A5	Security Misconfiguration	11	CWE-250	Execution with Unnecessary Privileges
			17	CWE-732	Incorrect Permission Assignment for Critical Resource
Flaw	A6	Sensitive Data Exposure	8	CWE-311	Missing Encryption of Sensitive Data
			19	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
			25	CWE-759	Use of a One-Way Hash without a Salt
Flaw	A7	Missing Function Level Access Control			
Bug	A8	Cross-Site Request Forgery (CSRF)	12	CWE-352	Cross-Site Request Forgery (CSRF)
Flaw	A9	Using Components with Known Vulnerabilities	16	CWE-829	Inclusion of Functionality from Untrusted Control Sphere

Table A-1 [50]

Cont.

Flaw	A10	Unvalidated Redirects and Forwards	22	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
Bug		Buffer Overflow	3	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
			18	CWE-676	Use of Potentially Dangerous Function
			20	CWE-131	Incorrect Calculation of Buffer Size
			23	CWE-134	Uncontrolled Format String
			24	CWE-190	Integer Overflow or Wraparound

Appendix B

Pre-Survey Questions

Please rate your level of knowledge or skills in the following areas using a scale of:

1(very low), 2(low), 3(medium), 4(high), 5(excellent)

	Rating (1-5)
1. Explain what an application logic flaw is	_____
2. Explain why it is difficult for an automated tool to find logic flaws	_____
3. Identify potential areas for logic flaws in HTTP interactions	_____
4. Critique code with logic flaws.	_____
5. Fix code with logic flaws to secure it.	_____

Post-Survey Questions

Please rate your level of knowledge or skills in the following areas using a scale of:

1(very low), 2(low), 3(medium), 4(high), 5(excellent)

	Rating (1-5)
1. Explain what an application logic flaw is	_____
2. Explain why it is difficult for an automated tool to find logic flaws	_____
3. Identify potential areas for logic flaws in HTTP interactions	_____
4. Critique code with logic flaws.	_____
5. Fix code with logic flaws to secure it.	_____

1. The case study was useful to help you understand the material.

a. Strongly Agree b. Agree c. Neither Agree or Disagree

d. Disagree e. Strongly Disagree

2. You were motivated to learn about application logic flaws.

a. Strongly Agree b. Agree c. Neither Agree or Disagree

d. Disagree e. Strongly Disagree

3. You enjoyed the case study.

a. Strongly Agree b. Agree c. Neither Agree or Disagree

d. Disagree e. Strongly Disagree

4. The learning objectives (questions 1-5) were met.

a. Strongly Agree b. Agree c. Neither Agree or Disagree

d. Disagree e. Strongly Disagree

5. The introduction and animation provided enough information for the discussion questions.

a. Strongly Agree b. Agree c. Neither Agree or Disagree

d. Disagree e. Strongly Disagree

6. The case study material was organized.

a. Strongly Agree b. Agree c. Neither Agree or Disagree

d. Disagree e. Strongly Disagree

7. The level of difficulty of this case study is appropriate.

a. Strongly Agree b. Agree c. Neither Agree or Disagree

d. Disagree e. Strongly Disagree

8. The case study helped you to develop problem solving and critical thinking skills, and ability to think independently.

a. Strongly Agree b. Agree c. Neither Agree or Disagree

d. Disagree e. Strongly Disagree

9. What was the most important thing you learned from this case study?

10. What problems did you encounter in completing this case study?

11. Provide any general comments about the case study.

Appendix C

Discussion Questions

Note:

Questions 3, 4, and 5 use the same example code. This should let the students try the manual code review method for finding logic flaws. All three questions should be given as a set.

Questions 7 and 8 use the same example code; only one should be given.

Notation used:

Unsigned argument: argument
Signed Argument: (ARGUMENT)
Attacker-accessible argument in red text
Attacker-accessible APIs in **black text**
Store arguments and code in blue text
Cashier arguments and code in green text

1. What is a logic flaw, and why are they difficult to identify?

2. Why is it NOT a security vulnerability that the price, payerID, result, sessionID, storeID, token, and PayPalExpress URL parameters from the Interspire and PayPal Express case are unsigned? Explain why for each variable.

3. Read through the following URLs representing HTTP interactions between three parties. Suggest potential security flaws for each of the arguments.

   The cart is the cart object stored by the Store. The IPN_Handler refers to the Instant Payment Notification API method used by the Store, where GoogleCheckout notifies the Store immediately after the user makes a payment. Refer to Section III. B 3) in *How to Shop for Free Online* for details about this case.

   1. Store.com/updateCart?price&sessionID
   2. Store.com/checkout?price
   3. Google.com/pay?(SESSIONID)&(CART)&(IPN_HANDLER)
   4. Store.com/(IPN_HANDLER)?(SESSIONID)&(STATUS)
   5. Order has been completed

4. Read through the following code and critique it. Refer to Section III. B 3) in *How to Shop for Free Online* for details about this case, and slides 5-10 of the PowerPoint.

1. Store.com/updateCart?myPrice&mySessionID

```
if (mySessionID < 0 || mySessionID >= currentSessionID)
        return; //sessionID is invalid
carts[mySessionID].price = myPrice;
```

2. Store.com/checkout?myPrice

```
createCart(myPrice)
        carts[currentSessionID].price = myPrice;
        currentSessionID++;
sign(currentSessionID-1);
sign(carts[currentSessionID-1]);
sign(IPN_HANDLER);
```

3. Google.com/pay?(SESSIONID)&(CART)&(IPN_HANDLER)

```
    verifySignature( (SESSIONID));
    verifySignature( (CART) );
    verifySignature( (IPN_HANDLER) );
    recordPayment( (CART).price, (SESSIONID) )
            payments[currentPaymentID].paymentAmount = (CART).price;
            payments[currentPaymentID].orderID = (SESSIONID);
```

4. Store.com/(IPN_HANDLER)?(SESSIONID)&(STATUS)

```
status = OK;
sign(status);
IPN_Handler( (SESSIONID), (STATUS) )
if ( (STATUS) == OK){
        if ( (SESSIONID)< 0 || (SESSIONID) >= currentSessionID)
                return; //(SESSIONID) is invalid

        createOrder(carts[(SESSIONID)].price, PAID, GoogleCheckout)
                orders[currentOrderID].orderID = currentOrderID;
                orders[currentOrderID].price = carts[(SESSIONID)].price;
                orders[currentOrderID].status = PAID;
                orders[currentOrderID].paymentType = GoogleCheckout;
        status = orders[currentOrderID].status;
        sign(status);
}
```

5. Order has been completed

5. Fix the code from Question 4 to better secure it. Refer to Section III. B 3) in *How to Shop for Free Online* for details about this case.

6. Propose methods to test for logic flaws (does not have to be automated).

7. Given the following code and variable table, trace through the variables at each stage.

   This is a replay attack, where the attacker gains access to a signed variable and can reuse it. They must pay for the first order, and leave the orderID argument in the URL empty. This means the orderID in the signed variable is empty.

   The flaw is that the store retrieves the orderID from the cookie, if the orderID in the signed variable is empty. The attacker can modify the orderID in the cookie to a new order, and complete additional orders at the same price as the first order, without paying for them.

   See the file: Question 7 – trace table.

8. The following URLs/code sequence is out of order. What is the correct sequence (give the page numbers)? Refer to section III B. 2) in *How to Shop for Free Online* for details about this case.

   This is a replay attack, where the attacker gains access to a signed variable and can reuse it. They must pay for the first order, and leave the orderID argument in the URL empty. This means the orderID in the signed variable is empty.

   The flaw is that the store retrieves the orderID from the cookie, if the orderID in the signed variable is empty. The attacker can modify the orderID in the cookie to a new order, and complete additional orders at the same price as the first order, without paying for them.

   See the file: Question 8 – out of order.

Appendix D

Table D-1

Pre-survey aggregate data

Learning Objective	**1**	**2**	**3**	**4**	**5**
1	3	4	5	5	5
2	3	4	6	5	3
3	10	7	6	8	9
4	2	1	1	0	1
5	0	2	0	0	0

Table D-2

Post-survey aggregate data for learning objectives

Learning Objective	**1**	**2**	**3**	**4**	**5**
1	0	0	0	0	1
2	1	1	1	2	0
3	4	7	7	9	9
4	8	6	6	3	4
5	1	0	0	0	0

Table D-3

Post-survey aggregate data for how much the students agree or disagree with these statements

	Strongly Agree	Agree	Neither Agree nor Disagree	Disagree	Strongly Disagree
The case study was useful to help you understand the material	4	7	1	2	0
You were motivated to learn about application logic flaws	4	4	5	1	0
You enjoyed the case study	4	5	3	2	0
The learning objectives were met	1	10	3	0	0
The introduction and animation provided enough information for the discussion questions	1	8	3	2	0
The case study material was organized	2	7	4	1	0
The level of difficulty of this case study is appropriate	2	5	4	3	0
The case study helped you to develop problem solving and critical thinking skills, and ability to think independently	3	6	5	0	0

Table D-4

Quiz Grades

100	97.5	90	85	82.5	80	75	45.9
11	1	1	1	1	2	3	1

Table D-5

Discussion Question Final Report Grades by Group

100%	95%	90%	85%	75%
2	2	1	1	1

Table D-6

Discussion Question Grades by Question

Question	1 (mid-term exam)	2	3	4	5	8
100%	12	7	3	4	5	5
75%	4		3	3		1
50%	5		1		2	1
25%						
0%						